Capturing Survey Client-side Paradata

Hueichun Peng and Jason Ostergren, University of Michigan Survey Research Center, United States

1. Abstract

Paradata that are captured during the survey process are a valuable source of information in helping us understand and improve the data collection process.

One of the advantages of using Blaise for survey data collection is the rich capture of paradata that is native to the application. Paradata which are linked directly to the administration of a survey instrument are collected automatically through the Blaise software (i.e., audit trail). The ADT file from Blaise 4 has been very valuable in understanding interviewer behavior. With the advent of Blaise IS and Blaise 5, we now are able to increase the richness of our paradata collection for understanding respondent behavior on web-SAQ (self-administered questionnaires) and/or mixed mode projects (i.e., interviewer and web-SAQ combined).

The team at the University of Michigan has implemented a process to capture paradata from the client-side using JavaScript. This client-side paradata (CSP) can capture movement within a page on the respondent's or interviewer's device like scrolling and mouse-clicks. As research interest has been strong in using these paradata to analyze respondents' behavior, we are working to capture more events like "loss of focus" and "geolocation". Furthermore, as more and more respondents start to use mobile device to take web surveys, we added a new library to capture the events specific to mobile device such as tapping, orientation change, pinch-in, etc. As the University of Michigan has been investigating using tablets (mobile) for CAPI interview due to cost and efficiency consideration, the capture of mobile paradata will be crucial for analyzing interviewer's behavior as well.

We will share what we have learned and the challenges we have considered, with the intent that other Blaise 5 users can also supplement their paradata capture.

2. Introduction

Paradata that are captured during the survey process are a valuable source of information. They help researchers better understand the respondents' behavior and their device. They also have the potential to improve the data collection process.

There are two main types of paradata collected in web surveys: server-side paradata and client-side paradata. The terms "server-side" and "client-side" are based on the technical concepts in web programming that the scripts or functions are primarily triggered and executed at the web server (server-side) or the web browser (client-side). Server-side paradata usually tracks the respondent's *visit* to the web page, capturing timestamps, the respondent's unique identifier, question items on the page, question answer provided when R moves to next page, browser type, user agent string (UAS), etc. Client-side paradata records the respondent's actions within a specific page by means of JavaScript. Generally it records the *timestamp* (timespan) together with the actions: for example, when the respondent clicks a radio button, enters some texts, unclicks the radio button, scrolls the browser window, etc.

These paradata are collected from web surveys deployed either as a web-SAQ (self-administered questionnaires) or an interviewer-administered SAQ.

Currently, the primary web survey platforms the Survey Research Center has been using include Blaise and Illume. For each, we have added extra scripts and functions to capture the server-side and client-side paradata.

With the ever increasing number of respondents taking the web surveys on mobile device and newer browsers equipped with HTML5 features, we decide to enhance our client-side paradata to include events specific to mobile platforms and other events that could provide more information about respondents' behavior and their device.

This paper focuses on the client-side paradata (CSP) enhancement we have worked on, mostly related to mobile platform and enhancements on various areas. It describes the proof of concept we tested experimentally, the implementation with Blaise-5 instruments, and other enhancements we have been working on. Detailed implementation of the last version of our CSP JavaScript functions in BlaiseIS can be found in the paper presented at the 2010 Blaise conference (Ostergren and Liu, 2010).

3. Events on Mobile device – a Proof of Concept

In 2015, we added a new JavaScript library of functions in a cross-campus web survey (on the Illume Web Survey platform) in addition to the original CSP JavaScript library we used to collect CSP. The new library of functions aims to capture new events mostly related to mobile platform, including the screen size at page-loading time, tapping, pinching (in and out) and pinch distance, and orientation change. The JavaScript functions are primarily based on the following JavaScript events:

- Window Screen size in pixels when page is loaded: Record the window.screen.height and window.screen.width.

- Tap: Add listener to capture the Touch (OnTouchStart) event. We identify Tap event when the touches.length =1

- Pinch: Add listener to capture the Touch (OnTouchStart and OnTouchEnd) event, and calculate the distance between the start point and end point

- Orientation change: The original orientation can be inferred from the starting screen size. Add listener to capture the window.orientation properly to identify the new orientation.

The new library *only* recorded specific events that mostly happen on mobile platform and did not record the basic events logged by our original library. Our plan was to compare the data from the two sources and merge them into one library after testing and validation. We decided to incorporate this testing into this project with cross-campus surveys for which the majority of the respondents are college students.

Below are examples of the raw data captured in comparison with the CSP captured from the original library of function.

- ***Example-1 from IPhone:***

    - CSP from the new library is as below.

      414,736;landscape,portrait,tapped,tapped,19.977213593219687,pinch_together,tapped,tapped,tapped,tapped,tapped,tapped,tapped,tapped,tapped,tapped,

      Data Notes:
        - *414,736 -> screen size in pixels.*
        - *landscape, portrait -> change of orientation*
        - *99999 pinch_together -> pinch together by the distance of 9999*
        - *tapped -> any touching event recorded*

    - The corresponding CSP from the original library is as below.

      ^t=40290:endScrollForCSP(x=false,y=true)^t=123116:Q39=4^t=956:endScrollForCSP(x=false,y=true)^t=1005:Q41=3^t=723:endScrollForCSP(x=false,y=true)^t=23

68:Q42=4^t=988:endScrollForCSP(x=false,y=true)^t=3685:Q68=1^t=23480:DATSTAT.NEXT=[CLICK]

Data Notes:

- *^t: milliseconds after page-load*
- *endScrollForDCP (x=false,y=true) -> scrolling action at either x or y direction*
- *Q39=4 -> Users enters response 4 for Question ID Q39*
- *DatStat.Next = [Click] -> Users click the Next button*

- ***Example 2 from Android (HTC One Max):***

  - CSP from the new library is as below.

    432,768;landscape,tapped,tapped,14.09574790846645,pinch_together,portrait,tapped,tapped,

    Data Notes: explained as the above

  - The corresponding CSP from the original library is as below.

    ^t=5218:endScrollForCSP(x=true,y=true)^t=1154:STUDQUES10A_2015=4^t=708:DATSTAT.NEXT=[CLICK]

    Data Notes: explained as the above

This experiment suggests that the new tracking data provide valuable information about our users' behavior on a Mobile OS. We decide to consolidate the 2 libraries by merging the mobile event listener into the original CSP JavaScript functions. We implement the new library to a production project that uses Blaise-5 Web Surveys.

4. Implementation on Blaise-5 Web Survey

The following is a discussion of the details of our Blaise 5 client-side paradata (CSP) implementation, from the use of the Data Entry API to the JavaScript we use. Generally speaking, there are three necessary parts to developing a scheme to capture CSP. These are:

1. Developing a JavaScript to capture the CSP on the client browser.

2. Developing a mechanism to save the captured CSP upon server contact during page change.

3. Developing a mechanism to update the captured CSP string with information about the current state that cannot be reproduced later (namely replacing temporary and possibly reused element identifiers from the html with related fieldnames, taking advantage of the server-side ability to do that translation).

The JavaScript we are using to collect Blaise 5 CSP is a further refinement of the script described in the IBUC 2010 paper by Jason Ostergren and Youhong Liu entitled "BlaiseIS Paradata." The main purpose of our CSP JavaScript is to capture keystrokes and mouse clicks on question elements such as text boxes and radio buttons, and to capture other actions that may take place in the intervening time, such as scrolling and pinch/ zoom. Capturing keystrokes and clicks may show, for example, that the user changed answers many times before leaving the page, possibly indicating a problem with the clarity of the question or task. Incidentally, that information is also available to some extent in the keystroke-level paradata provided with Blaise 5 out-of-the-box. Capturing scroll activity and zooming gives us an idea of how often the content exceeded the viewable area and how much effort was required for the user to absorb the content of the full page, alerting us to whether it was formatted for efficient viewing. This type of information is not provided in the out-of-the-box paradata. The inclusion of both types in our CSP script (keystroke-and-click-type CSP and scroll-and-zoom-type CSP), gives us more information about the order of events on a page, since one can determine which typing or clicking events took place before a scroll/zoom event and which took place after.

In terms of technical details, we looked at third-party libraries to capture touch paradata like pinch-zoom (for now we have settled on “hammer.js”), because detecting such activity requires handling of spatial and temporal tolerances for what constitutes such an event that are beyond what we currently want to invest in the effort. We discovered some technical limitations along the way, such as the fact that Firefox touch events are presently disabled by default in Windows (discovered because the script is being developed on a Surface Book). Because Windows-based touch devices have not been commonly used to complete surveys, we are not attempting to address this problem at the present time, but this points to the fact that unexpected platform and browser considerations can still cause unexpected results (and potentially hinder development).

It may also be worth noting that, while it might have proved easier to write this script with the aid of the JQuery library, we have run into a few problems that convinced us to stick with standard JavaScript. This is because Blaise 5 is currently using a somewhat older version of JQuery, and we have run into a few conflicts in other scripts when we do not match that version of JQuery exactly. We have, however, integrated our script into the Require.js framework that Blaise 5 currently uses, which is the reason for the define statement at the beginning of the script.

The description of the custom Data Entry API code (hereafter: router) implementation provided here is based on current (as of August 2016) Data Entry API architecture for ASP, but we know that the upcoming release will include a new architecture meant to supersede it. While the principle will remain the same, the details will no doubt differ.

The router implementation begins with attaching some code to events raised by the IDataEntryControllerAsp implementation. These are:

ControlFactory.OnCreateDataFieldInputControl, ControlFactory.OnCreateCategoryInputControl, BeforeExecuteActions. Other code needs to be added to provide a hidden field on the page in which to store the captured CSP, but that is not specific to Blaise 5.

An important part of the process is to update the captured CSP with identifiers that are meaningful for later analysis. The element ids connected with the client-side events that the JavaScript is capturing are no longer meaningful once the page changes (the next page may have identical ids referring to different question elements). Therefore, in the server-side code, which has access to the Blaise API, there must be code which translates the temporary identifiers used on the page into something like fieldnames. This can be done by cataloguing the fieldnames associated with the temporary identifiers while those html elements are being generated. The ControlFactory.OnCreateCategoryInputControl event provides the opportunity to do this for controls like radio buttons associated with the categories in codeframes. Here is a code snippet illustrating how we handle this:

Listing 1. ControlFactory.OnCreateCategoryInputControl Source Code

```
            ICategory category = e.DataObject;
            string fieldName = category.Field.Name;
            if (category.Field.ValueType.DataType ==
StatNeth.Blaise.API.DataRecord.DataType.Set)
            {
                fieldName += "-" + category.Code;
            }
            string clientID = definition.ID;
            StoreClientIds(clientID, fieldName);
```

For ControlFactory.OnCreateDataFieldInputControl the code is slightly different to account for textbox naming conventions (note that the “_mask” naming is for custom controls developed by SRC, which shows how the system can be extended if needed).

Listing 2. ControlFactory.OnCreateDataFieldInputControl Source Code

```
            string fieldName = field.Name;
            string clientID = "";
            string definitionID = definition.ID;
            if (!string.IsNullOrEmpty(rawMask))
            {
                clientID = definitionID + "_mask";
            }
            else
            {
                clientID = definitionID + "_tb";
            }
            StoreClientIds(clientID, fieldName);
```

The StoreClientIds function called at the end of the above two snippets saves these pairs until the captured CSP is returned to be stored in the instrument database. We use the BeforeExecuteActions event to handle updating these ids and saving the CSP, like so:

Listing 3. BeforeExecuteActions Source Code

```
        if (_hiddenField != null)
        {
            IRouteItem paraDataField =
_depcontroller.Page.RouteItems.GetItem("ClientSideParadata");
            if (paraDataField != null)
            {
                string csp = _hiddenField.Value;
                foreach (KeyValuePair<string, string> clientIdentifier in
_clientIdentifiers)
                {
                    csp = csp.Replace(":" + clientIdentifier.Key, ":" +
clientIdentifier.Value);
                }
                paraDataField.Value.Assign(csp);
                _clientIdentifiers = new Dictionary<string, string>();
            }
        }
```

Note that the field accessed using Page.RouteItems.GetItem("ClientSideParadata") is one that we set as a Field Reference in the Resource Database. The above function takes the captured CSP (accessible via the _hiddenField variable we add to the page) and replaces the temporary identifiers with fieldnames from the collection constructed as described above. This modified string is then assigned to the Field Reference. After it is assigned back, the Blaise 5 source code handles concatenating this CSP to that collected from previous pages, stored permanently in a field called ClientSideParadataStore as seen in this snippet of Blaise 5 source code we have placed near the beginning of our datamodel:

Listing 4. ClientSideParadataStore Source Code

```
       ClientSideParadata.keep
       ClientSideParadataStore.keep
       IF ClientSideParadata <> EMPTY THEN
             ClientSideParadataStore := ClientSideParadataStore +
ClientSideParadata
             ClientSideParadata := EMPTY
       ENDIF
```

To be clear, we add an Open field named ClientSideParadataStore to each of our instruments and this one field captures all the CSP for each case (which can be lengthy). It is extracted normally with the data like any other Open field we use. Here is an example of the data contained in this field after a few pages:

Listing 4. CSP Data

```
^t=5:onload-SecA.StartInterview.A006_=screen-1500x1000,client-
1374x712,dt=2016-08-30T18:13:16.609Z
^t=2396:SecA.StartInterview.A006_=1[ENTR]
^t=0:onload-SecA.StartInterview.A007TRAlive_A=screen-1500x1000,client-
1374x712,dt=2016-08-30T18:13:20.363Z
^t=3293:PinchApart(Scale=3.7391,Duration=1003)
^t=73:Scroll(x=true,y=true)
^t=2934:SecA.StartInterview.A007TRAlive_A=1
^t=1601:PinchTogether(Scale=0.2272,Duration=474)
^t=1238:Scroll(x=true,y=true)
^t=-124:SecA.StartInterview.A007TRAlive_A=[ENTR]
^t=0:onload-SecA.StartInterview.A002_IwBegin=screen-1500x1000,client-
1374x712,dt=2016-08-30T18:13:30.211Z
^t=2884:SecA.StartInterview.A002_IwBegin=1[ENTR]
^t=2:onload-SecA.StartInterview.A155_SelfPrxy=screen-1500x1000,client-
1374x712,dt=2016-08-30T18:13:34.123Z
^t=6062:Help=[CLICK]
^t=2581:SecA.StartInterview.A155_SelfPrxy=1[ENTR]
```

Each line begins with the delimiter "^t=" and the number of milliseconds since the last event, or since the script loaded if the page just changed. Each page change is indicated by the presence of the text "onload-". In this example, you can see the respondent typing numeric answers and pressing enter to advance the survey which has one question per page. There are also examples of "pinch zoom" with information on the scale and duration of the pinch event. Incidentally, pinch zoom is the only touch event that we capture which is fully working for us right now. Note that the pinch event by definition causes normal scroll activity to register in our script as well, and this appears in the CSP output even though the user would not think of the pinch as scrolling. We may remove this possibly redundant information later, because we generally try to avoid capturing things we don't need in order to keep the size of the CSP string transmitted to the server as short as possible, not to mention limiting the size of the fully concatenated CSP string stored in the ClientSideParadataStore field. Finally, there is an example of a click on a help button near the end of the above CSP.

5. More to listen and more to capture

CSP has been providing important and interesting information about respondents' behavior on the screen (like the events on the browser) as well as some environment information about the respondents' machine (like screen dimension and browser type). Researchers are becoming increasingly interested in new areas, properties or events that provide information about the respondents' behavior or their environment. Although we see an increasing number of respondents using mobile devices to take web surveys, we also find a higher percentage of break-offs with mobile platforms. (Couper and Peterson, 2016) Although we implemented efforts to make the display mobile friendly, like changing the font size, color and/or changing table questions to item-by-item questions vertically stacked, it seems that respondents still have higher break-off rates when using a mobile device. There is considerable literature addressing questionnaire design issues related to this point. To help understand the causes of this issue, the CSP captured on mobile platforms might help us learn more about respondents' behavior and their challenges on a mobile browser, Thereby informing both questionnaire design and usability on mobile platform.

Besides logging events specific to mobile platforms, we also recognize that other attributes or behaviors might be requested because they are of interest to research communities, so we have been continuously working on the enhancement of our CSP library. We also recognize that some information from CSP has the potential to improve our operational efficiency and efficacy as well (like in a CATI Web Mode).

- **GPS location:** Record navigator.geolocation.getCurrentPosition (which is only available with the html5 compliant browsers) to know the latitude and longitude of the device.

- For security reasons, when a web page tries to access location information on the user's machine, the user is notified and asked to grant permission. Each browser has its own policies and methods for requesting this permission. For example, some will work only if the device turns on the location service and explicitly allow for user GPS satellites. If the device does not turn on the location service or the user does not *agree* with the service, no warning will display and no data will be recorded.

  For implementation purposes, the pop-up confirmation/agreement window to collect GPS information from the device has raised concern that this might discourage respondents from taking the web survey. On the other hand, if the web survey is collected by field interviewers as a data entry mechanism, this feature might be helpful for operational and QC purposes because it helps to identify the physical location where the survey is conducted.

- **Coordinates of the object that being clicked on or touched on:** event.clientX or event.touches.screenX. Record coordinates of the mouse pointer when the mouse button is clicked on an element or the coordinate of the touch point relative to the screen, not including any scroll offset.

  Recording the coordinates of the objects might provide information about the exact display on the various platforms. Although we generally use responsive design to achieve optimal display on different browsers, there may be still differences among different mobile devices. Also, recording this might help us create a click-map and provide info about usability issues.

  Also, the coordinates of the clicked spots unveil some interesting aspects of users' behavior. For a simple radio button question, users could be clicking on the radio button itself, or the users might be clicking *around* the control (like the label of a radio button).

  There is a variety of coordinates that can be tracked with either JavaScript or JavaScript library like JQuery. For example, the coordinates of the monitor screen, the fully rendered content area in the browser, or the content area (the viewport) of the browser window. These attributes might become more complex and have different implications on mobile platforms. For example, when we zoom in on a mobile device, what do we wish to capture -- the position of the current browser screen, namely, the visual viewport (the part of the page that's currently shown on-screen) or the corresponding location coordinates relative to the original page layout, namely, the layout viewport (synonym for a full page rendered on a desktop browser)? All the different coordinates might have different *meanings* or might address different research purposes.

- **Window Focus or Blurred:** Add Event Listener to the event of "focus" and "blur" to track if respondents stay focused on the web survey pages.

- **Double-Tap:** On some mobile devices, double tapping triggers zooming effect, or hover-click event. We use the tapping events to capture this if the tapping occurs within 3 milliseconds.

CSP from our new testing SCP library is as below.

- **Example 1 from IPhone:**

  - dimensions:414,736;orientation:portrait; ^t:243629,coord:(42.27419841289512,-83.74521005579236);^t:277411,tap:(134,411);^t:277916,direction:together,distance:1;

^t:278765,tap:(546,860); ^t:279355,tap:(566,1537);^t:279926,tap:(513,2014); ^t:281373,tap:(299,1978);^t:281486,direction:apart,distance:-26; ^t:282104,tap:(267,2196);

  - dimensions:414,736;orientation:portrait;^t:387666,coord:(-27.487006101743173,152.99031924516342); ^t:390988,orientation:landscape; ^t:392899,orientation:portrait; ^t:394585,orientation:landscape; ^t:395964,orientation:portrait; ^t:402344,orientation:landscape; ^t:404388,orientation:portrait; ^t:407339,window:blurred; ^t:414152,window:focused;

  - *Data Notes: please see below explanations in Example 2 from Android*

- **Example 2 from Android:**

  - dimensions:432,768;orientation:portrait;^t:20871,coord:(42.2743974,-83.7451881); ^t:23494,orientation:portrait; ^t:25507,tap:(497,364);

  - dimensions:432,768;orientation:portrait;^t:263992,coord:(42.2743974,-83.7451881); ^t:274153,tap:(329,616);

  - dimensions:768,432;orientation:landscape; ^t:478316,coord:(42.2679279,83.7419255); ^t:482056,orientation:landscape;^t:488314,tap:(293,757);^t:488896,doubletap:(310,669); ^t:489925,tap:(148,654);

  - *Data Notes:*
    - *dimensions: x, y -> the screen size in pixels*
    - *orientation: landscape or portrait -> orientation and orientation change*
    - *coord: (x,y) -> the GPS location*
    - *^t: x -> action occurs at x milliseconds after page load*
    - *window: blurred -> window focus leaves the current page*
    - *window: focused -> window focus returns to the current page*
    - *tap: (x,y) -> touch the screen at the coordinates of (x,y)*
    - *direction: together, distance x -> pinch together by the distance of x*
    - *direction: apart, distance: y -> pinch apart by distance of y*
    - *doubletap: (x,y) -> tab on the coordinates of (x,y) twice within 3 milliseconds*

- **Items to work on**

  - **Swipe and Scroll:** we are currently working on capture the swipe-scroll events on mobile platform. We plan to use the touch event with the direction of movement for this.

  - **Zoom in or out after double tapping**

6. Implementation and Deployment

We have been continuously working to enhance our CSP JavaScript function library. However, there is still significant work and decisions to make regarding the specific items to include and record in a production environment.

We need perform load testing to make sure the *listening and recording* will not slow down the page or interfere with any other JavaScript function on the page. For example, in a few tests we conducted, we found the time to record GPS location seems higher on a mobile device than that on a regular PC. As we do this GPS recording as the 1st step in our JavaScript function, it is not clear if the delay is caused by the device, by the slow response time from the browser get the GPS location, or the respondent’s response time to the warning message with the GPS recording. These questions need further investigation before we can implement the functions in a production setting.

Technically, we are trying to expand our horizon and sharpen our *ears* to listen to more events that happen on the page, browser or the respondents’ environment. These events will be expected to further inform research objectives and improve operational efficiency.

7. References

Ostergren, Jason, Liu, Youhong. 2010. BlaiseIS Paradata. *Presented at the 13th International Blaise Users Group Conference, Baltimore, USA*

Couper, Mick P., Peterson, Gregg. 2016. Why Do Web Surveys Take Longer on Smartphones? *Social Science Computer Review 2016.*